SGX-mTCP: executing the user-level mTCP stack inside an Intel SGX enclave

Keita Aihara, Pierre-Louis Aublin and Kenji Kono
Keio University, Japan

***Abstract*—Trusted Execution Environments allow an application to run securely even in an untrusted environment such as the Internet. However, existing approaches still rely on an untrusted network stack for their communications. Unfortunately, bugs or security breaches can jeopardize the communication.**

To tackle this problem we describe *SGX-mTCP*, a secure TCP/IP stack which leverages the Intel SGX Trusted Execution Environment. SGX-mTCP executes the TCP stack alongisde the application inside an SGX enclave to provide integrity and confidentiality guarantees of the TCP state.

We implement SGX-mTCP and evaluate it with microbenchmarks, the Lighttpd web server, and the Memcached key-value store. Our experimental results demonstrate that SGX-mTCP protects the TCP stack of cloud applications with moderate performance overheads.

***Index Terms*—Trusted Execution Environment, TCP stack**

I. Introduction

We describe SGX-mTCP[1], a port of the mTCP user-level TCP stack to Intel SGX. SGX-mTCP replaces the untrusted network stack with a trusted network stack and can be either used by standalone or libraryOS-based secure applications.

SGX-mTCP is composed of several components depicted in Fig. 1. IP packets are received and sent by the network card via a fast *user-space packet I/O library*, that does not involve the operating system or network drivers during network communications, forwards incoming and outgoing IP packets between the NIC and the TCP stack. It does not need to inspect or modify the packets. The IP packets are processed by a *secure network stack*, executing inside a TEE. The network stack reassembles the IP packets into the TCP stream when receiving data, and creates the TCP stream and IP packets when sending data. The application data is exchanged between the application and the network stack via the TCP buffers.

In our design the application executes within the same TEE as the secure TCP stack. Existing techniques [1] can further isolate the TCP stack and the application, yet at the expense of performance.

To minimize the amount of code that has to be trusted, the packet I/O library executes outside of the TEE. This, however, does not affect the security of SGX-mTCP.

II. SGX-mTCP

A. Secure TCP stack

The core software component of SGX-mTCP is a secure network stack. It executes inside a TEE and is in charge of handling the TCP stream and IP packets. In the case of a multithreaded application, a secure network stack thread is created for each application thread that does network communication. While executing inside the same address space, the secure TCP stack threads do not share any data. For performance reasons, the application and secure network stack execute in a single process and inside the same TEE. Their interaction consists of normal function calls via a well-defined API that is similar to the BSD sockets API (see §III). The network can be accessed by multiple applications concurrently. Each application, and each application thread inside the same application, uses a different secure network stack instance.

As mentioned previously, the packet I/O library is executing outside a TEE. This is because executing it inside a TEE does not increase the security of SGX-mTCP, but simply puts more code and data into the TCB. The role of the packet I/O library is only to copy IP packets between the NIC and the secure stack. Further, even if executed inside a TEE, the malicious host can still send packets to the NIC via DMA operations (they will eventually be discarded by the secure network card thanks to SGX-mTCP authentication).

The execution flow of the secure TCP stack is as follows: it first retrieves IP packets from the I/O library, copies them into the TEE and checks for their integrity, possibly discarding invalid packets. It can then proceed to the reconstruction of the TCP stream. When the application issues a call to read data, data is copied by the secure network stack from the TCP buffers to the application buffers (and decrypted in case the application uses TLS). Sending data across the network is performed in a similar way.

Besides processing TCP packets, the secure stack is also in charge of processing ARP packets. By removing the processing of these packets from the operating system to the secure stack SGX-mTCP prevents the malicious host operating system from launching powerful ARP attacks.

B. Attestation and provisioning

The attestation mechanism (i) ensure that each component of SGX-mTCP is correctly initialized, with the TEE executing the correct code on its trusted hardware; and (ii) provision secrets to the different components.

The design of the attestation mechanism of SGX-mTCP is based on the Intel SGX attestation mechanism. The attestation and provisioning processes rely on a trusted third-party. This can be a separate entity similar to existing certificate authorities. When started, the TEE creates a *quote*, a report of

[1]The source code of SGX-mTCP is available at `https://github.com/sslab-keio/sgx-mtcp`.

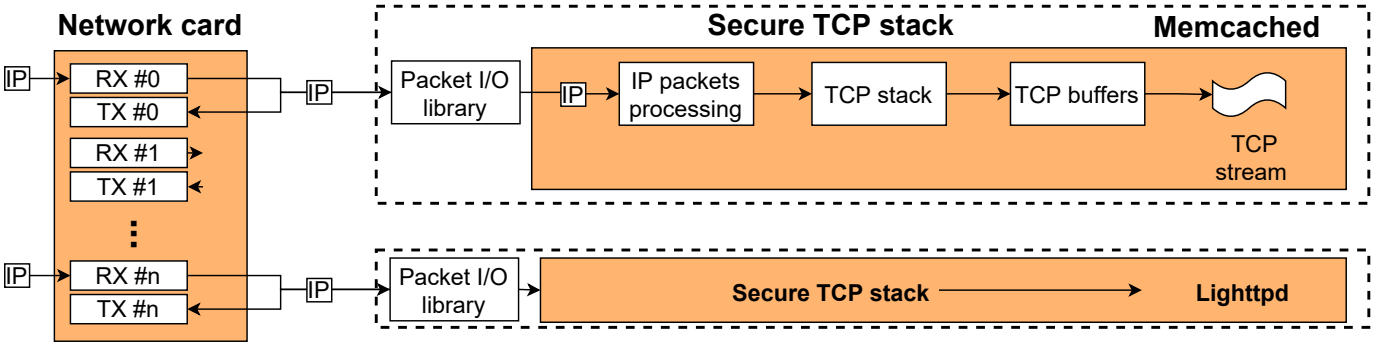


Figure 1: SGX-mTCP architecture with 2 applications: Memcached and Lighttpd. `IP` represents an IP packet; the orange colour represents a TEE. The dashed box represents a single address-space (process).

its memory content and hardware characteristics. It is signed using a cryptographic key known only by the trusted hardware. This report proves that the application executes the correct code on real trusted hardware. It can be either sent to another TEE executing on the same machine or to the trusted third-party, The former case is called *local attestation* while the latter case is called *remote attestation*.

III. Implementation

We ported the mTCP [2] user-level high performance TCP/IP stack to Intel SGX and use DPDK [3] for the user-level packet I/O library. mTCP differs from the BSD socket API in several ways: it (i) implements a per-core accept queue design, meaning that multiple cores can concurrently listen on the same socket; (ii) does not share sockets nor data structures across multiple cores, which improves concurrency and scalability; (iii) associates one mTCP thread, communicating with DPDK, with each application thread; and (iv) avoids expensive system calls to the kernel by running entirely in user-space.

A. Intel SGX

Intel SGX [4] is a new set of instructions present on Intel processors since 2015. It provides confidentiality and integrity guarantees even in the presence of a powerful attacker that controls both the hardware and software stacks (with the exception of the CPU package).

Intel SGX implements Trusted Execution Environments called *enclaves*, which are a special secure execution mode. The memory of applications running inside an enclave is stored inside a secure area called the *Enclave Page Cache* (EPC). On the most recent hardware the EPC size is limited to 256 MB, shared between all the enclaves of the system. As the EPC also contains SGX metadata, only around 188 MB is available to applications. Using more memory starts an expensive paging mechanism [5]. To provide confidentiality and integrity property, the enclave memory is transparently encrypted and its integrity is verified in hardware. Furthermore, while an enclave can access both trusted and untrusted memory, the application can only access untrusted memory.

Enclaves are accessed via a well-defined interface, composed of *enclave calls* (*ecalls*) and *outside calls* (*ocalls*). Executing an *ecall* enters the enclave and changes the execution mode from untrusted to trusted. Similarly, executing an *ocall* changes the execution mode from trusted to untrusted. Intel SGX adds additional checks to *ecalls* and *ocalls* to prevent attacks and leakage of secrets. The direct consequence is that enclave transitions are costly: the authors of sgx-perf [6] observed that an enclave transition takes $\approx 13,100$ cycles.

SGX-mTCP uses the attestation mechanism of provided by Intel SGX. It can be used by applications to prove to a third-party the authenticity of their enclave and the system on which they run [7]. This mechanism gives to clients the guarantee that they are communicating with a secure service.

B. Secure network stack

mTCP uses several timers, for example to check for time-outs. Access to a fast and trusted time in Intel SGX is only possible starting with SGX v2 [8]. Our implementation of SGX-mTCP currently relies on the untrusted time given by the untrusted operating system: SGX-mTCP retrieves the current time upon its regular *ocall* to DPDK to check for new incoming packets, thus avoiding the need for making another *ocall* and not affecting the performance. To the best of our knowledge machines that can both combine an SGX v2-capable processor and a NIC supporting DPDK are unfortunately not available yet.

The secure network stack interface is composed of 40 *ocalls*. They are used for accessing DPDK (15) and the standard library (25). We added 561 lines of code to mTCP and created an enclave wrapper (to handle *ocalls*) composed of $2,200$ lines of boiler-plate code. Given that our application executes inside an enclave, the number of *ecalls* depends on the application.

C. Performance optimisation

The performance of Intel SGX applications is limited by two factors: (i) the amount of memory used by the application; and (ii) enclave transitions.

To achieve the best performance each TCP buffer size is set to 82 MB. As a result our implementation requires 164MB of memory per core. Due to the small EPC size, we allocate the TCP buffers outside of the enclave. As it is often the case with current cloud-based applications, we assume the application uses a secure communication protocol such as

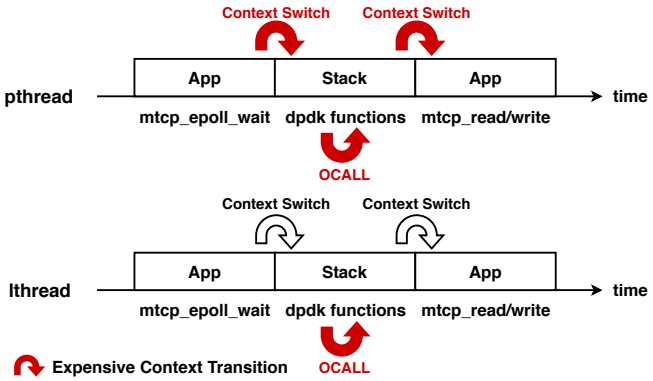


Figure 2: Design comparison between pthread and lthread

TLS or IPSec [9], [10]. Consequently, this does not affect the confidentiality nor integrity of the communications. With this optimisation the secure network stack fits in the EPC as it uses only 16 MB of enclave memory.

We used the sgx-perf profiler [6] to pinpoint performance bottlenecks. Our optimisations fall into 3 categories: (i) merge *ecalls* and *ocalls* to reduce the number of enclave transitions; (ii) move DPDK code inside the enclave to avoid *ocalls*; and (iii) user-level threading.

Merge calls. When possible, we merged *ecalls* and *ocalls* to reduce the number of enclave transitions. For example, the original mTCP implementation gets the current time (`gettimeofday()`) before packet processing (`dpdk_send_pkts()` and `dpdk_recv_pkts()`). With SGX-mTCP's secure network stack these three functions are *ocalls*. By merging these three *ocalls* into a single one we were able to reduce enclave transitions in the main loop of the secure stack thread by two third.

Move code. To further reduce the number of *ocalls* we moved several functions of DPDK into the enclave. In particular we moved the `get_rptr()` and `get_wptr()` that are called to retrieve, in the DPDK buffers, respectively the location of the next packets that can be read and the location where to write the next packets that have to be sent.

User-level threading. mTCP executes the application thread and corresponding mTCP thread on the same core to benefit from cache sharing and improve the performance. Unfortunately, the thread context switch overhead of Intel SGX is very high, making it difficult to offer acceptable performance. For this reason, we leverage user-level threading, with the `lthread` library. Fig. 2 shows the design comparison between `pthread` and `lthread`. Context switch of user-level threading happens in the enclave. It removes costly thread context transitions and subsequently improves the overall performance.

IV. Evaluation

Our evaluation first shows that SGX-mTCP is effective against a malicious host OS trying to access the network (§ IV-A). Then, after detailing our experimental setup (§ IV-B), it shows that SGX-mTCP exhibits a small performance overhead using three applications: the Memcached key-value store (§ IV-C), the Lighttpd web server (§ IV-D). Finally, we present a fine-grained analysis of the performance of SGX-mTCP using micro-benchmarks (§ IV-E), the impact of our performance optimisations (§ IV-F) and the scalability of SGX-mTCP (§ IV-G).

A. Security discussion

In this section we list attacks on SGX-mTCP. For each attack we show how SGX-mTCP prevents it, guaranteeing protection from an untrusted host OS.

TCP stream tampering. An attacker could modify the content of the TCP buffers stored in untrusted memory. Given that the content is encrypted and integrity-protected, this attack is detected by SGX-mTCP and the application. Note that this attack is specific to our implementation due to the limited EPC size.

Enclave interface attacks. An attacker could try to access the enclave secrets by manipulating the arguments and returned values of *ecalls* and *ocalls* [11], [12]. To reduce the attack surface we harden the enclave interface with additional checks on the values. Specifically, (i) we limit the number of pointers used by the interface functions, preferring passing data by value rather than reference; (ii) SGX-mTCP checks the origin of pointers and aborts if it detects that an argument of an interface function references enclave memory; (iii) SGX-mTCP checks and limits the size of data structures passed by reference, preventing buffer overflow attacks.

B. Experimental set-up

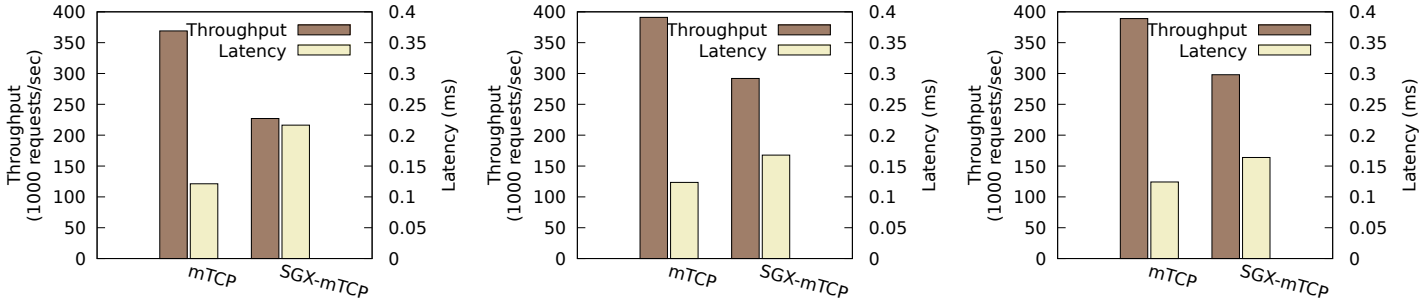
All the experiments are run on an SGX-capable machine composed of a 6-cores Intel Core i5-8500 at 3GHz (no hyperthreading) equipped with 16GB of RAM and running the Intel SGX SDK v2.5. This processor EPC size is 128MB, leaving only 90MB for secure applications. The client machine is a 6-cores Intel Xeon X5650 at 2.67GHz with 8GB of RAM. They both use a 10GbE NIC that supports DPDK and run Ubuntu 18.04.02 LTS with the Linux kernel 4.15.

Both the applications and secure stack execute inside an enclave. Unless otherwise noted, applications are executed on all the CPU cores and their data fits inside the EPC.

C. Memcached key-value store

We ported Memcached to Intel SGX and modified it to be able to run it with mTCP. This requires two major changes: (i) replacing calls to the socket API to calls to the mTCP API; and (ii) removing calls to the `libevent` library, calls that are replaced by the `epoll` system of mTCP. Our modifications, inspired by [13], have required 373 new lines of code.

We measure the performance with the YCSB benchmark [14], with values of 1 kB with uniform key access distribution, for the following workloads: YCSB A (50% get; 50% update), YCSB B (95% get; 5% update) and YCSB C (100% get). Other YCSB benchmarks would either provide similar results (YCSB D is similar to YCSB B) or are CPU



(a) YCSB A. (b) YCSB B. (c) YCSB C.

Figure 3: YCSB performance of Memcached.

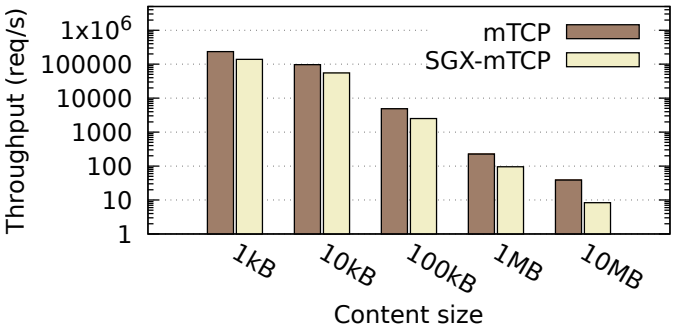


Figure 4: Lighttpd throughput with different web page sizes.

intensive and put less stress on the TCP stack (YCSB E and YCSB F). Two configurations are analysed: (i) with mTCP (no SGX); and (ii) with SGX-mTCP.

The single-core performance results are reported in Fig. 3. Executing Memcached inside an enclave shows a performance overhead of 23%–38%. This overhead is consistent with other papers: the authors of [15] report a 45% performance overhead while Weisse et al. [16] observed an overhead of 80% that they could reduce to 40% after optimisations. Memcached with SCONE [5] reports a throughput of 0.6x–1.2x compared to native execution. To achieve this result SCONE leverages hyperthreading, which could be leveraged by an attacker to tamper with the enclave security [17]. Note that our implementation of Memcached has not been optimised to run inside an SGX enclave. In particular, updating a key-value pair requires to take a lock, which can require an enclave transition, in turns increasing the number of thread context switches and degrading the performance as the number of concurrent threads increases.

D. Lighttpd web server performance

To assess the performance of SGX-mTCP with the Lighttpd web server [18], that we have ported to Intel SGX, we measure the maximum performance when accessing web pages of different sizes, from 1 kB to 10 MB. An analysis of the size of the main web page (including their associated scripts and images) of the Alexa top 500 websites [19] revealed that less than 0.8% of the web pages have a size of 1 kB or less, 50% of the web pages have a size of less than 229 kB, 90% of the web pages have a size of less than 3.4 MB, and the maximal web page size is 10 MB.

Results are reported in Fig. 4. The overhead of SGX-mTCP is between 42% and 78%. This is in line with the overhead shown in [16]. This overhead is due to the numerous enclave transitions needed by Lighttpd, mainly to access the web pages stored on the file system, which explains why it increases as the web page size increases.

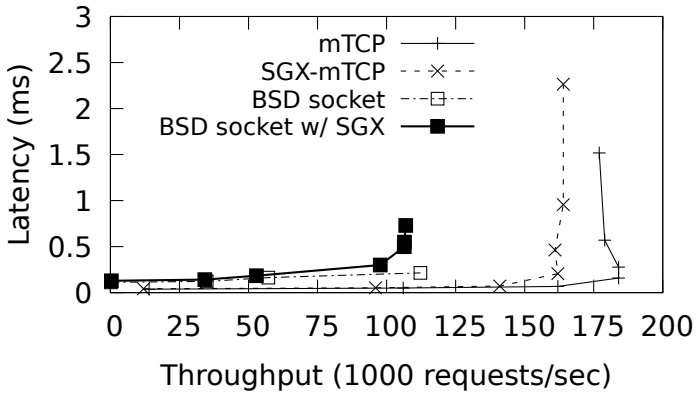
E. Microbenchmark performance

The microbenchmark consists of a simple client-server application. A set of clients exchanges fixed-size packets with the server application. Clients send one packet per TCP connection, meaning that they need to open a new TCP connection for each packet. As a result the benchmark shows the cost of sending and receiving TCP packets as well as establishing and closing TCP sessions. We monitor the maximal rate at which they exchange packets as well as the latency.

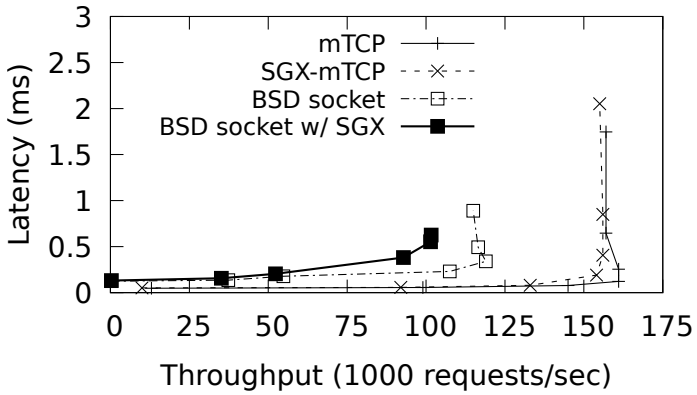
Fig. 5 presents the evolution of the throughput and latency in two different configurations: (i) 1 byte requests and replies (Fig. 5a); and (ii) 1 kB requests and replies (Fig. 5b). We report four configurations: (i) the unsecure `mTCP` network stack as our baseline; (ii) `SGX-mTCP`, where the microbenchmark and network stack execute inside an Intel SGX enclave; (iii) a benchmark using the BSD socket API; and (iv) the same benchmark but with the server running inside an SGX enclave. In all cases the server application is running on one core to show the single-CPU performance.

Running mTCP inside an enclave decreases the performance by less than 11%. For example, with 1 byte requests and replies, the maximal throughput decreases from 184,000 requests/sec to 164,000 requests/sec. In the case of 1 byte requests and replies, the maximal throughput decreases from 184,000 requests/sec to 156,000 requests/sec.

In another experiment we compare the performance of SGX-mTCP with the standard BSD socket API included in the Linux kernel. They are denoted by the `BSD socket` and `BSD socket w/ SGX` lines in Fig. 5. The maximal performance achieved is at most 118,900 requests/sec, with 1 kB requests and replies, which is 15% lower than SGX-mTCP. We can also



(a) 1 byte requests and replies.



(b) 1 kB requests and replies.

Figure 5: Throughput vs latency of mTCP, SGX-mTCP and the BSD socket API.

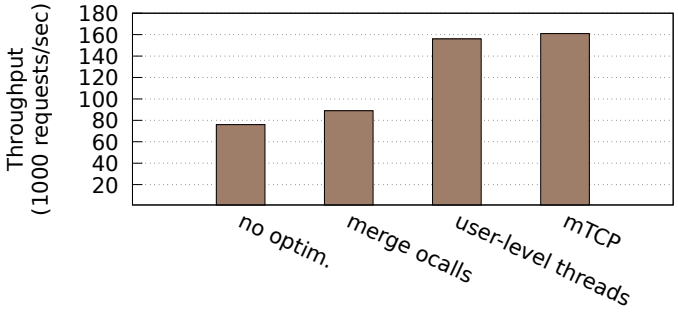


Figure 6: secure stack performance optimisations.

notice that the SGX version has a 15% overhead compared to the non-SGX version, providing at most 101,600 requests/sec. Similar observations can be made for 1 byte requests and replies. This further motivates the need for SGX-mTCP, which brings better performance and better security compared to the network stack offered by the operating system.

F. Performance optimisations

In this section we evaluate the impact of the different performance optimisations presented in § III-C.

Fig. 6 presents the maximal throughput observed after applying each optimisation with the microbenchmark and requests and replies of 1 kB. First of all, SGX-mTCP throughput without any optimisation is as low as 76,000 requests/sec. When merging *ocalls* and moving code when accessing DPDK functions, the throughput increases by 17%, to 89,000 requests/sec. Leveraging user-level threading increases the performance by 75%, up to 156,000 requests/sec. This optimisation reduces the context switch overhead when scheduling two different SGX threads on the same core. The takeaway lesson of performance optimisation with Intel SGX is that thread context switches and enclave transitions are very expensive and should be avoided.

#Threads	Relative throughput	
	mTCP	SGX-mTCP
1	1.0	1.0
2	2.0	1.8
4	3.4	3.0
6	3.7	3.7

Table I: Relative throughput of mTCP, single-enclave and SGX-mTCP as the number of threads increases.

G. Scalability

In this section we are interested in the scalability of SGX-mTCP as the number of cores it utilizes increases.

We measure the scalability of the secure stack component of SGX-mTCP as the number of threads increases with the microbenchmark and 1 byte requests and replies.

The results are shown in Tab. I. As can be observed, both mTCP and SGX-mTCP scale in a similar way, with a throughput improvement when using 6 cores of x3.7. This is because threads in mTCP and SGX-mTCP do not share any data structure and can run in parallel without requiring any synchronisation.

V. Related work

Researchers have already considered the execution of network functionalities inside an SGX enclave: Shieldbox [20] executes the Click modular router [21] inside SCONE [5] and uses DPDK for fast user-level packet processing. In a similar way, Endbox [22] executes a VPN and the Click modular router inside an SGX enclave to provide scalable middlebox functionalities at the client. These systems are complementary to SGX-mTCP.

Systems such as Haven [23], Graphene-SGX [24] or SGX-LKL [25] provide applications with essential functionalities by implementing an entire libraryOS [26] inside the TEE, including the TCP stack. However, they still rely on the host OS network stack for network communications. By porting

SGX-mTCP to these systems this dependency can be removed and stronger security guarantees are obtained.

Several systems that improve the performance and security of the network stack have been proposed in the past. IX [27], Arrakis [28] and Shinjuku [29] use virtualisation techniques to separate the network processing from the rest of the kernel; Hodor [30] provides secure and fast kernel-bypass I/O access via compiler instrumentation. Contrarily to these systems, SGX-mTCP protects network communications even in the presence of a powerful adversary who controls the operating system and tries to launch attacks on the network.

SR-IOV [31] is a special feature of network cards that allows the virtualisation of a network interface. Using SR-IOV, a guest OS can bypass the host OS when accessing the network device. Nonetheless, the host OS is in charge of setting up the network card. An untrusted host OS is able to inspect and modify the network traffic destined to the guest OS. SGX-mTCP can leverage SR-IOV to offer better performance to trusted applications executing inside a virtual machine.

TownCrier [32] is a datafeed system for smart contracts that bridges HTTPS-based websites and the blockchain. It leverages Intel SGX to securely process TLS traffic and HTTP requests. TownCrier can make use of SGX-mTCP in order to improve the protection of the untrusted host operating system.

VI. Conclusions

This paper presents SGX-mTCP, a port of the mTCP user-level TCP stack to Intel SGX. We implemented a prototype of SGX-mTCP that uses Intel SGX and demonstrated, using two applications (web server and key-value store) as well as microbenchmarks, that SGX-mTCP incurs moderate performance overhead while protecting the integrity and confidentiality of the TCP state.

Acknowledgements

This work was supported by JST CREST Grant Number JPMJCR19F3, Japan.

References

[1] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel sgx," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 955–970.

[2] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a highly scalable user-level TCP stack for multicore systems," in *NSDI '14*, 2014, pp. 489–502.

[3] T. L. Foundation, "Dpdk project," https://www.dpdk.org/, 2019.

[4] Intel, "Software Guard Extensions Programming Reference, Ref. 329298-002US," https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth *et al.*, "SCONE: Secure linux containers with Intel SGX," in *OSDI '16*, 2016.

[6] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A performance analysis tool for intel sgx enclaves," in *Proceedings of the 19th International Middleware Conference*. ACM, 2018, pp. 201–213.

[7] S. P. Johnson, V. R. Scarlata, C. V. Rozas, E. Brickell, and F. McKeen, "Intel sgx: Epid provisioning and attestation services," *Intel*, 2016.

[8] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3D: System Programming Guide, Part 4*, September 2016.

[9] OpenSSL Software Foundation, "OpenSSL," https://www.openssl.org/, 2016.

[10] C. R. Davis, *IPSec: Securing VPNs*. McGraw-Hill Professional, 2001.

[11] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," in *ASPLOS*, vol. 13, 2013, pp. 253–264.

[12] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei, "Coin attacks: On insecurity of enclave untrusted interfaces in sgx," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 971–985.

[13] S. Thongprasit, V. Visoottiviseth, and R. Takano, "Toward fast and scalable key-value stores based on user space tcp/ip stack," in *Proceedings of the Asian Internet Engineering Conference*, ser. AINTEC '15. New York, NY, USA: ACM, 2015, pp. 40–47. [Online]. Available: http://doi.acm.org/10.1145/2837030.2837036

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[15] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic application partitioning for intel sgx." Santa Clara, CA, USA: USENIX, 07/2017 2017.

[16] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 81–93, 2017.

[17] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*, 2017, pp. 1–6.

[18] J. Kneschke, "Lighttpd," https://www.lighttpd.net/, 2003.

[19] Alexa, "Top Internet sites," https://www.alexa.com/topsites, Oct. 2019.

[20] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "Shieldbox: Secure middleboxes using shielded execution," in *Proceedings of the Symposium on SDN Research*. ACM, 2018, p. 2.

[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[22] D. Goltzsche, S. Rüsch, M. Nieke, S. Vaucher, N. Weichbrodt, V. Schiavoni, P.-L. Aublin, P. Costa, C. Fetzer, P. Felber, P. Pietzuch, and R. Kapitza, "EndBox: Scalable middlebox functions using client-side trusted execution," in *Dependable Systems and Networks (DSN), 2018 48th Annual IEEE/IFIP International Conference on*. Luxembourg, Luxembourg: IEEE, 2018.

[23] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," ser. OSDI, 2014.

[24] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library os for unmodified applications on sgx," 2017.

[25] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "Sgx-lkl: Securing the host os interface for trusted execution," 2019.

[26] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 291–304.

[27] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *OSDI 14*, 2014, pp. 49–65.

[28] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 4, p. 11, 2016.

[29] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for μsecond-scale tail latency," in *NSDI 19*, 2019, pp. 345–360.

[30] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC '19)*, 2019.

[31] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471–1480, 2012.

[32] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*, 2016, pp. 270–282.